Algoritmos sobre Grafos

Ingeniería en Computación e Informática

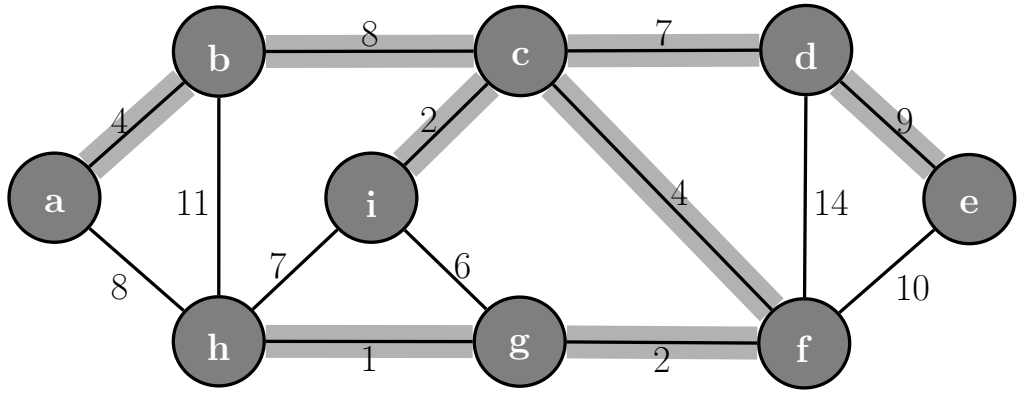
Tabla de contenidos

Árboles de Cobertura Mínima

MST: *Minimum Spanning Trees*

Definición

Un árbol de cobertura para un grafo no dirigido conexo $G = (V, E)$ es un subgrafo de G que es un árbol no dirigido y contiene todos los vértices de G. En un grafo ponderado $G = (V, E, W)$, el peso de un subgrafo es la suma de los pesos de las aristas incluidas en ese subgrafo. Un árbol cobertura mínima para un grafo ponderado es un árbol cobertura cuyo peso es mínimo.



Árboles de Cobertura Mínima

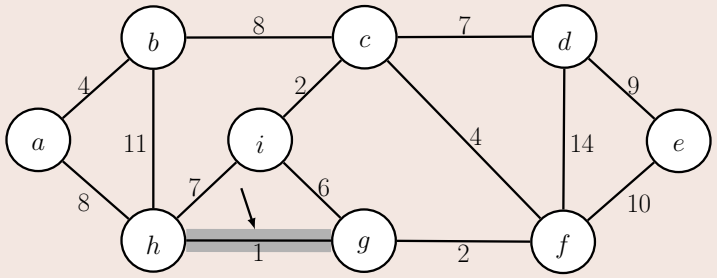
MST: *Minimum Spanning Trees*

- Se examinarán dos algoritmos para resolver el problema MST: el algoritmo de Kruskal y el algoritmo de Prim.
- Kruskal tiene una complejidad:
  - $O(|E| \log |E|)$ con una implementación en una estructura simple.
  - $O(|E| \log |V|)$ mediante estructura de datos sobre conjuntos disjuntos (*disjoint-set*).
- En tanto, el algoritmo de Prim se ejecuta en tiempo:
  - $O(|V|^2)$ en una matriz.
  - $O(|E| \log |V|)$ en *heap* binario con una lista de adyacencia.
  - $O(|E| + |V| \log |V|)$ en *heap* de Fibonacci con una lista de adyacencia.

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

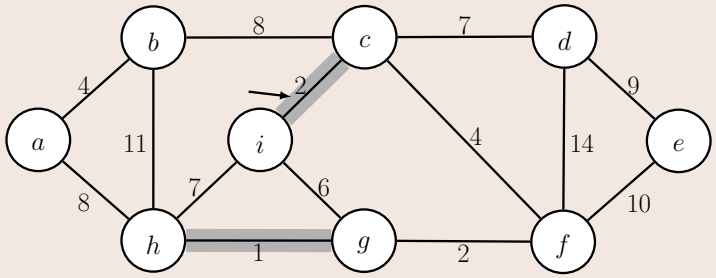
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

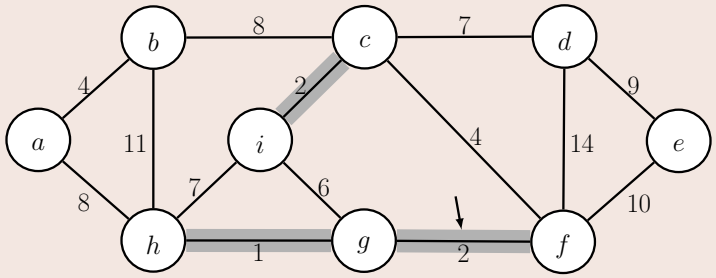
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

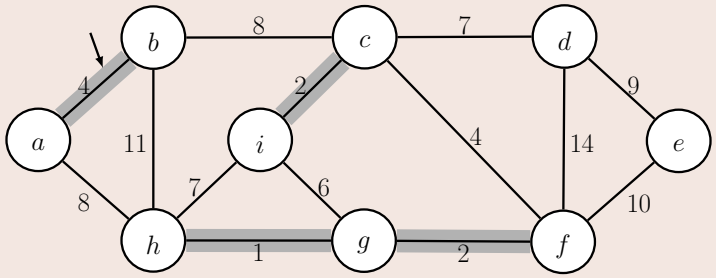
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1:  A ← ∅.
2:  for cada vértice u ∈ G.V do
3:      MakeSet(u)
4:  end for
5:  Ordenar las aristas de G.E en orden creciente de peso w.
6:  for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:      if Find(u) ≠ Find(v) then
8:          A ← A ∪ {(u, v)}
9:          Union(u, v)
10:     end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

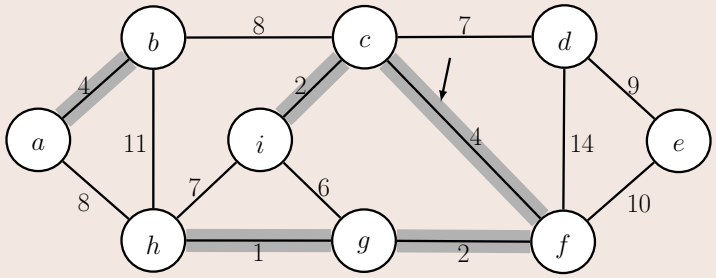
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

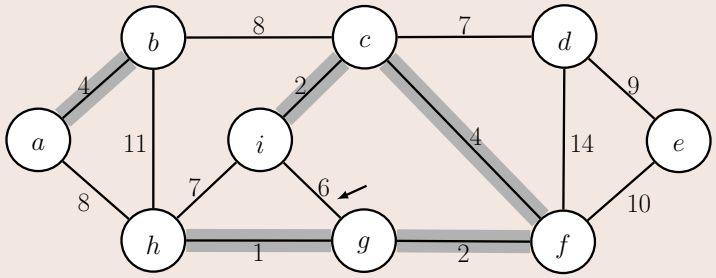
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

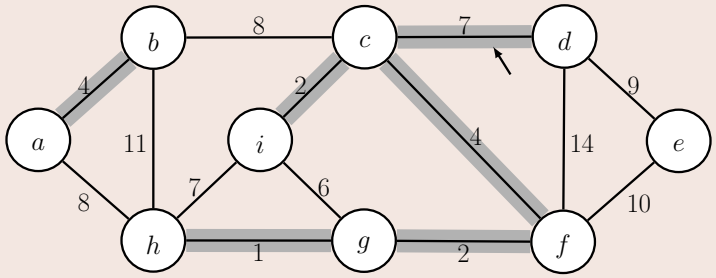
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

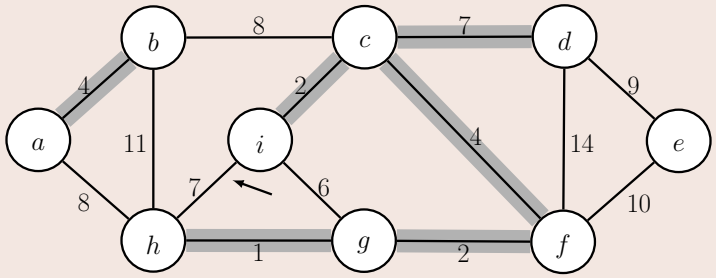
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

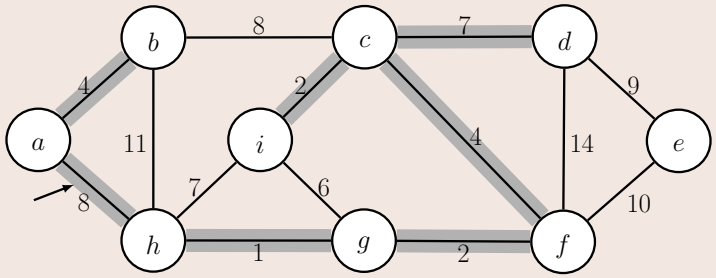
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

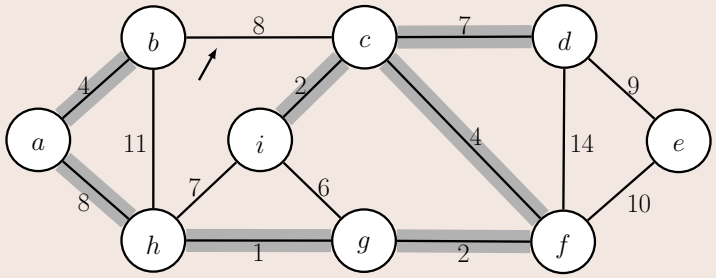
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

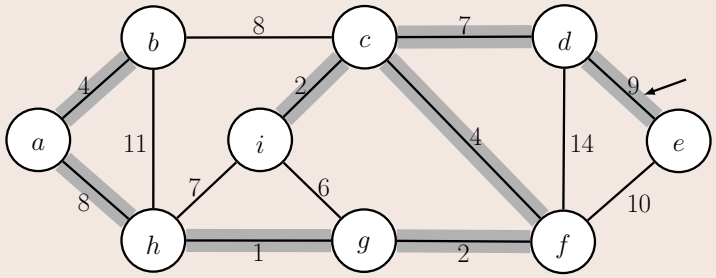
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

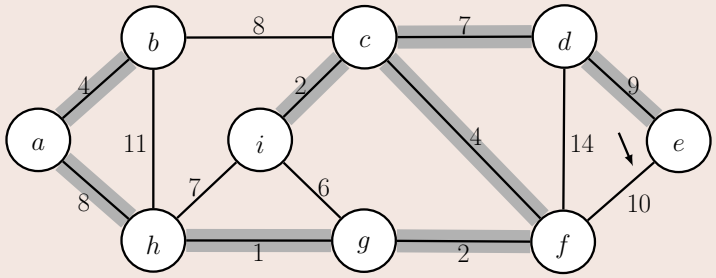
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

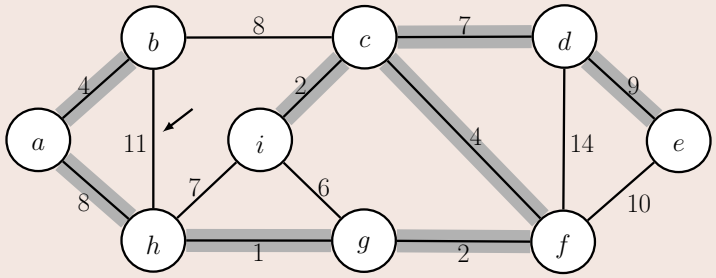
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

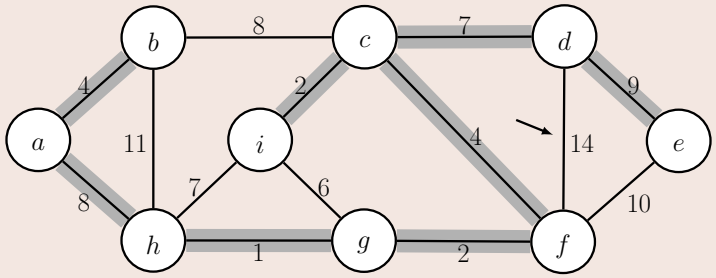
Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

Algoritmo de Kruskal

Implementado con *disjoint-set*



Algorithm KRUSKAL

Require: G: Grafo, w: pesos de las aristas
Ensure: A: aristas del MST.

```
1: A ← ∅.
2: for cada vértice u ∈ G.V do
3:     MakeSet(u)
4: end for
5: Ordenar las aristas de G.E en orden creciente de peso w.
6: for cada arista (u, v) ∈ G.E, tomada en orden creciente de peso do
7:     if Find(u) ≠ Find(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u, v)
10:    end if
11: end for
```

Árboles de Cobertura Mínima

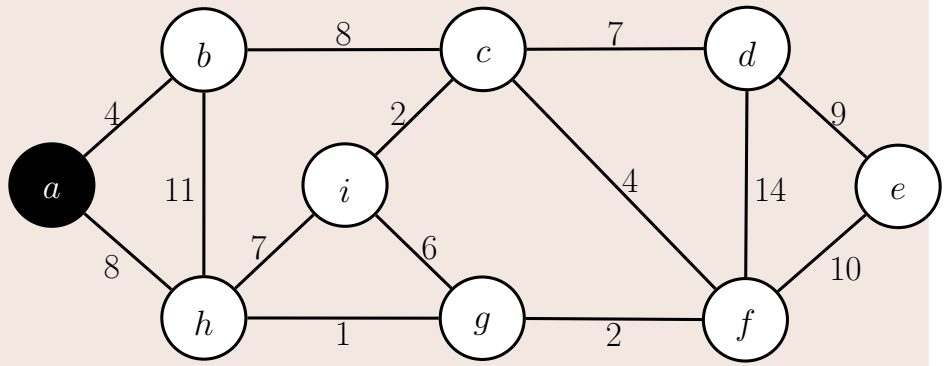
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Universidad Andrés Bello

Árboles de Cobertura Mínima

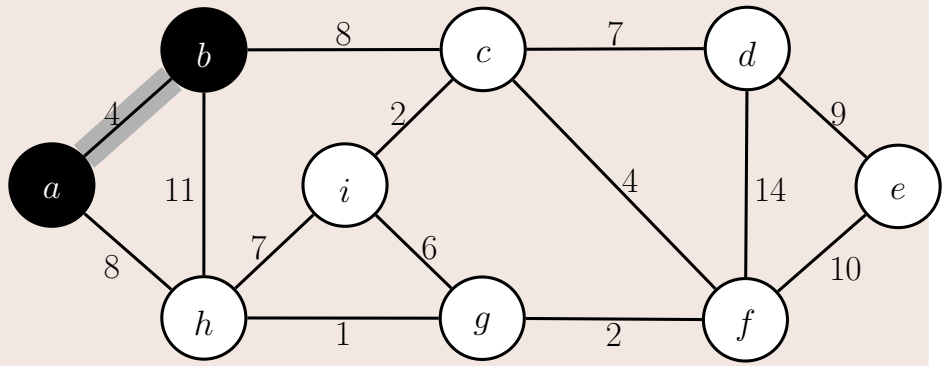
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Universidad Andrés Bello

Árboles de Cobertura Mínima

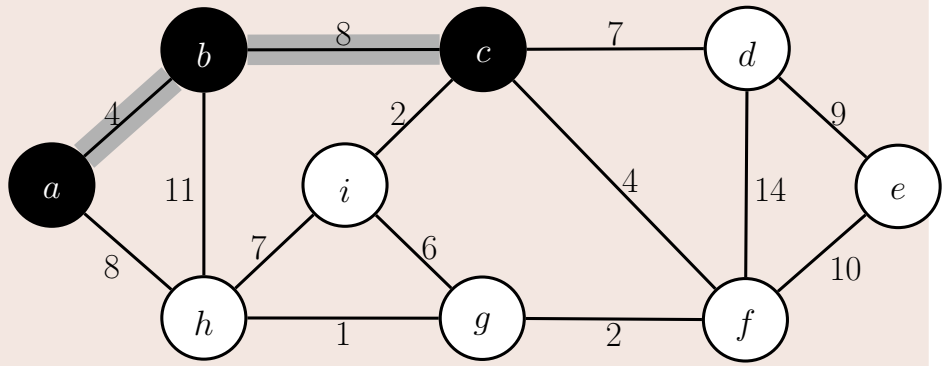
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Árboles de Cobertura Mínima

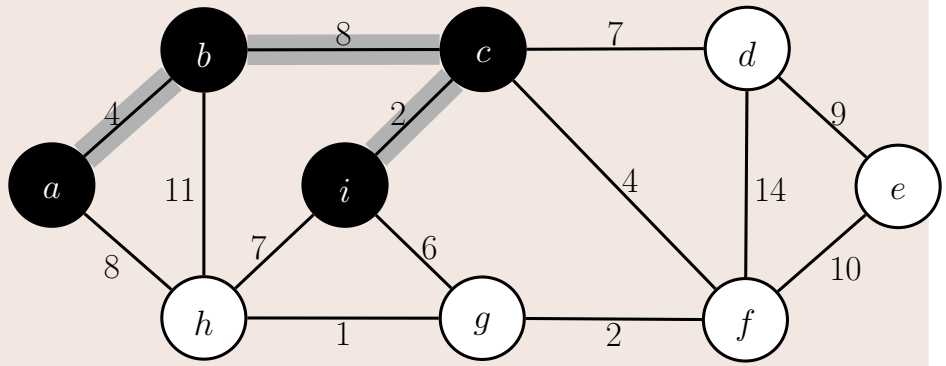
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Árboles de Cobertura Mínima

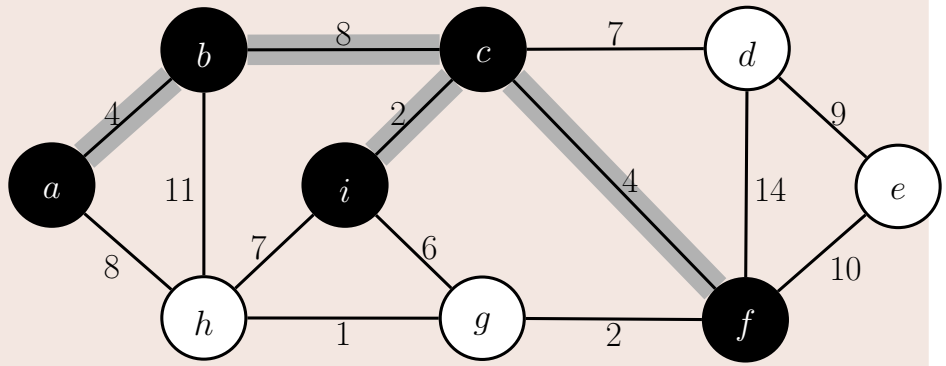
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Árboles de Cobertura Mínima

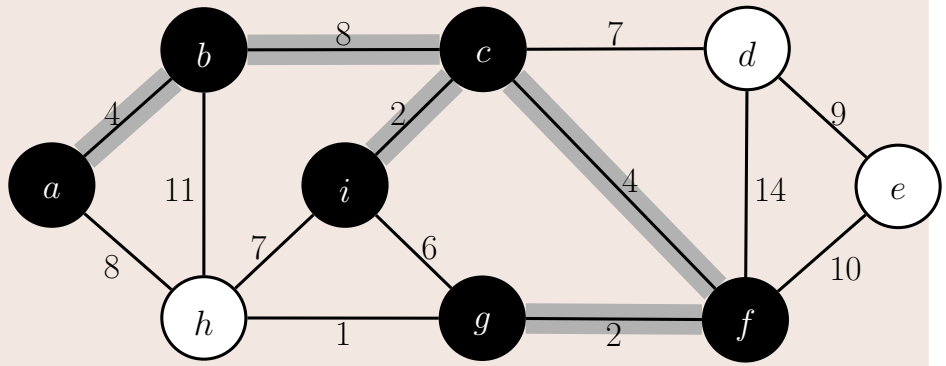
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Árboles de Cobertura Mínima

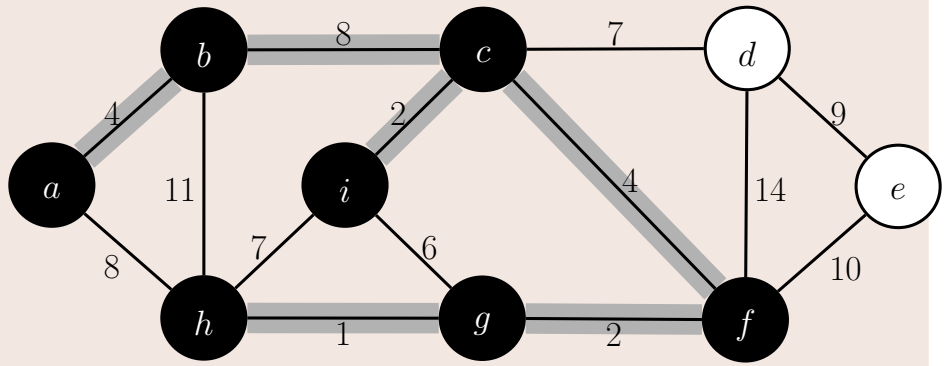
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Universidad Andrés Bello

Árboles de Cobertura Mínima

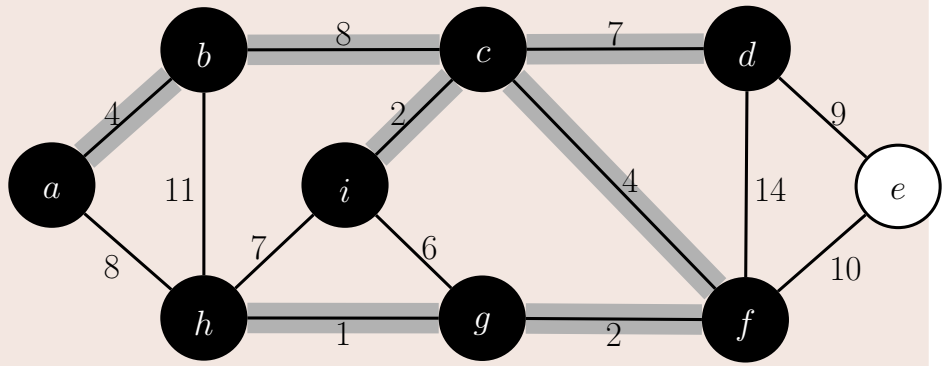
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Árboles de Cobertura Mínima

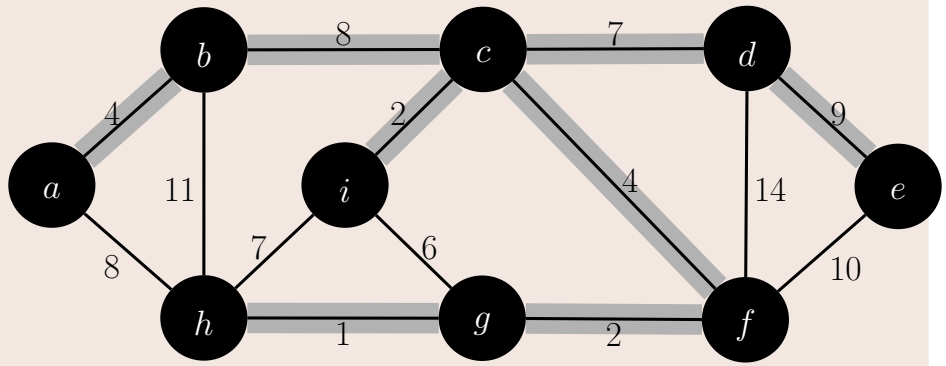
Algoritmo de Prim

Implementado con *heap* binario

Algorithm PRIM

Require: G: Grafo, w: pesos de las aristas, u: vértice inicial.
Ensure: G Grafo con key y π en vértices.

```
for cada v ∈ G.V do
    v.key ← ∞
    v.π ← NULL
end for
u.key ← 0
Q ← G.V
while Q ≠ ∅ do
    u ← EXTRAER-MIN(Q)
    for cada v ∈ G.Adyacente(u) do
        if v ∈ Q y w(u, v) < v.key then
            v.π ← u
            v.key ← w(u, v)
        end if
    end for
end while
```



Universidad Andrés Bello

Caminos Mínimos

Definición

Dado un grafo dirigido con peso $G = (V, E)$, se tiene un función de peso $w : E \to \Re$ que asigna valores reales a las aristas. El peso de un camino $p = \langle v_0, v_1, \ldots, v_k \rangle$ es la suma de los pesos de las aristas que lo constituyen:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Se define el peso del camino más corto de u a v como:

$$\delta(p) = \begin{cases} min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & si\ existe\ un\ camino\ de\ u\ a\ v \\ \infty & caso\ contrario \end{cases}$$

Caminos Mínimos

Variantes

El problema tiene variantes:

- **El problema de los caminos más cortos desde un origen**: en el cual tenemos que encontrar los caminos más cortos de un vértice origen v a todos los demás vértices del grafo.

- **El problema de los caminos más cortos con un destino**: en el cual tenemos que encontrar los caminos más cortos desde todos los vértices del grafo a un único vértice destino, esto puede ser reducido al problema anterior invirtiendo el orden.

- **El problema de los caminos más cortos entre todos los pares de vértices**: el cual tenemos que encontrar los caminos más cortos entre cada par de vértices $(v\ ,\ v')$ en el grafo.

Caminos Mínimos

El camino más corto desde un origen a muchos destinos

- Dado un grafo $G = (V, E)$ se desea encontrar un camino más corto desde un vértice origen dado $s \in V$ a cada vértice $v \in V$.
- Algunas veces, en los problemas de caminos más cortos existen aristas cuyo peso es negativo.
- Si el grafo $G = (V, E)$ contiene ciclos con peso no negativo alcanzable desde el origen s, entonces para todo $v \in V$, el peso del camino más corto $\delta(sv)$ continua bien definido, incluso si tiene valor negativo.

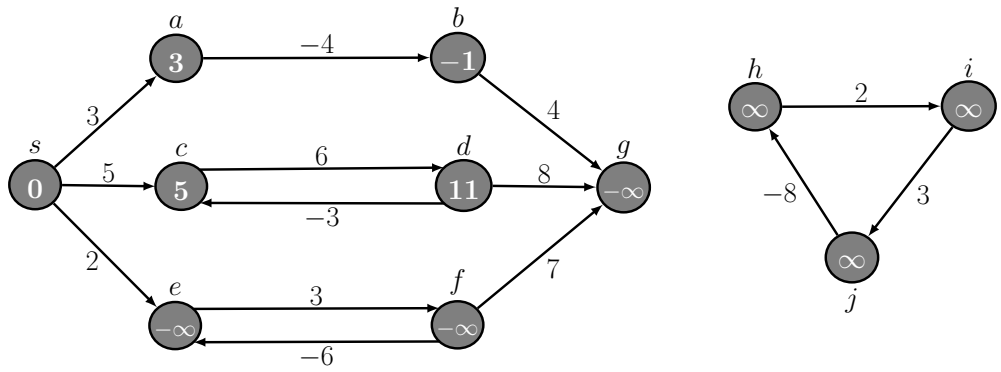
Caminos Mínimos

El camino más corto desde un origen a muchos destinos

- Sin embargo, si existe un ciclo con peso negativo alcanzable desde s, el peso del camino más corto no está bien definido.
- Ningún camino de s a un vértice en el ciclo puede ser un camino más corto (siempre habrá un camino "más corto").
- Si existe un ciclo con peso negativo en algún camino de s a v, se define $\delta(s, v) = -\infty$.

Caminos Mínimos

El camino más corto desde un origen a muchos destinos



- Como los vértices e y f forman un ciclo con peso negativo alcanzable desde s, ellos tienen pesos de caminos más cortos de $-\infty$.
- Como el vértice g es alcanzable desde un vértice cuyo peso de camino más corto es $-\infty$, entonces el también tiene peso de camino más corto de $-\infty$.
- Los vértices h, i y j no son alcanzables desde s, luego su peso de camino más corto es ∞ (aún cuando están en un ciclo de peso negativo).

Caminos Mínimos

Relajación

- Para cada vértice $v \in V$ se tiene un atributo $v.d$, el cual es un límite superior en el peso del camino más corto desde el origen s a v. El valor $v.d$ se denomina el *peso estimado del camino más corto*.

- Se inicializan los pesos estimados de los caminos más cortos y predecesores con el Algoritmo en $\Theta(|V|)$.

Algorithm INICIALIZAR-ÚNICO-ORIGEN

Require: G: Grafo, s: vértice origen.
1: **for** cada vértice $v \in G.V$ **do**
2: $v.d \leftarrow \infty$
3: $v.\pi \leftarrow NULL$
4: **end for**
5: $s.d \leftarrow 0$

Caminos Mínimos

Relajación

- El proceso de relajación de una arista (u, v) consiste en examinar si se puede mejorar el camino más corto a v pasando por u y, si es el caso, actualizar $v.d$ y $v.\pi$.
- La relajación puede decrementar el valor del peso estimado del camino más corto $v.d$ y actualizar el campo $v.\pi$ del predecesor de v.
- El Algoritmo realiza una relajación de la arista (u, v) en $O(1)$.

Algorithm RELAJACIÓN

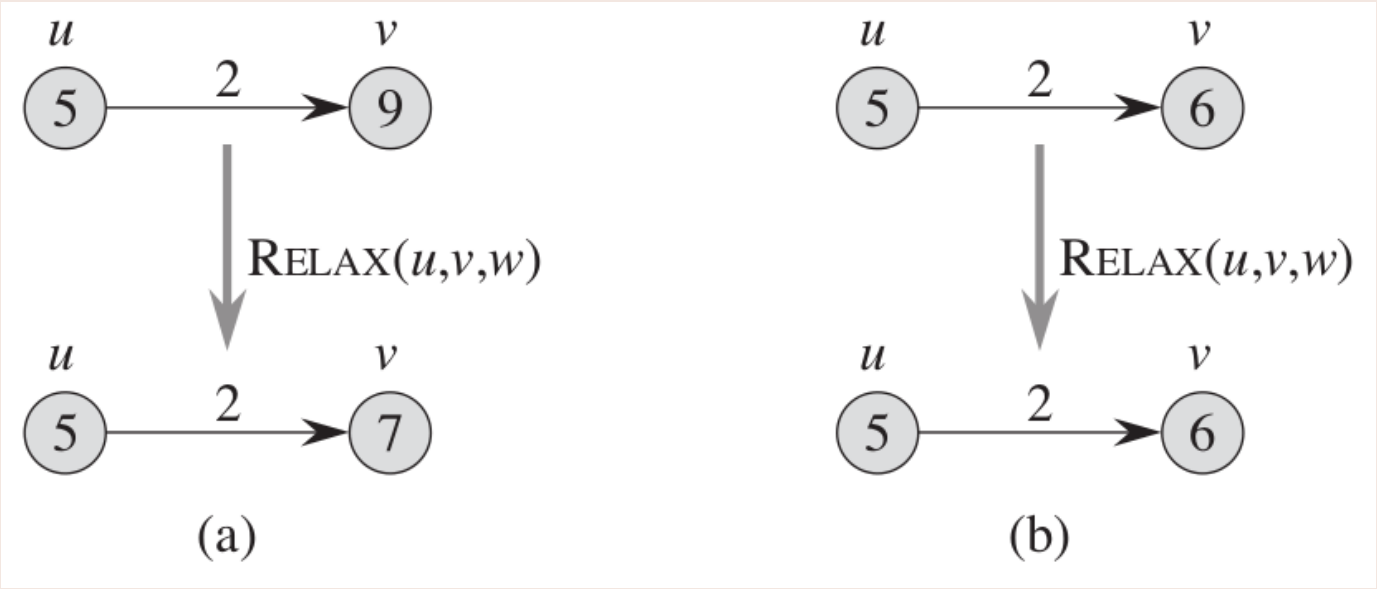
Require: u: arista $\in (u, v)$, v: arista $\in (u, v)$, w: peso de (u, v).

1: **if** $v.d > u.d + w(u, v)$ **then**
2: $\quad v.d \leftarrow u.d + w(u, v)$
3: $\quad v.\pi \leftarrow u$
4: **end if**

Caminos Mínimos

Relajación

Ejemplo



La relajación de una arista en a) donde el peso estimado del camino más corto decrece y en b) no hay cambios en el peso estimado.

Caminos Mínimos

Algoritmo Bellman-Ford

- El algoritmo Bellman-Ford resuelve el problema de caminos más cortos desde un origen a muchos destinos en el caso general en el cual los pesos de las aristas pueden ser negativos.
- Dado un grafo dirigido con peso $G = (V, E)$ con origen s y función de peso $w : E \to \Re$, el algoritmo retorna un valor booleano que indica si existe o no un ciclo de peso negativo que es alcanzable desde el origen.
- Si no hay un ciclo de peso negativo, el algoritmo produce los caminos más cortos y sus pesos.

Caminos Mínimos

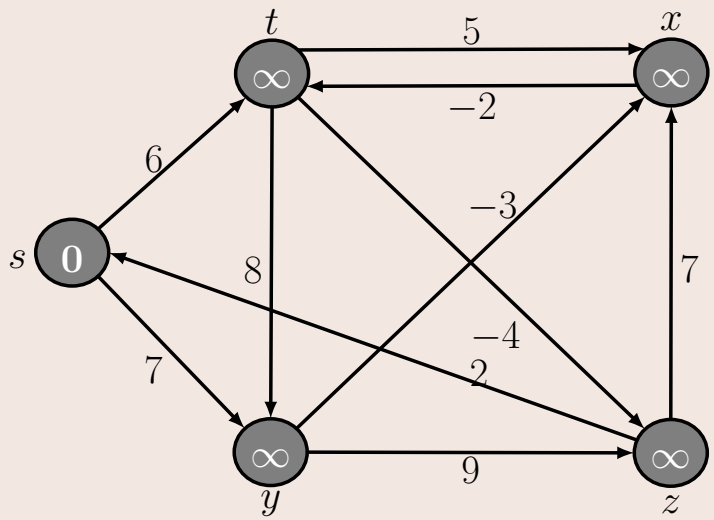
Algoritmo Bellman-Ford

Implementado con *heap* binario

Algorithm Bellman-Ford

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
 1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
 2: for i ← 1 to |G.V| − 1 do
 3:     for cada arista (u, v) ∈ G.E do
 4:         RELAJACIÓN(u, v, w)
 5:     end for
 6: end for
 7: for cada arista (u, v) ∈ G.E do
 8:     if v.d > u.d + w(u, v) then
 9:         return false
10:     end if
11: end for
12: return true
```



Caminos Mínimos

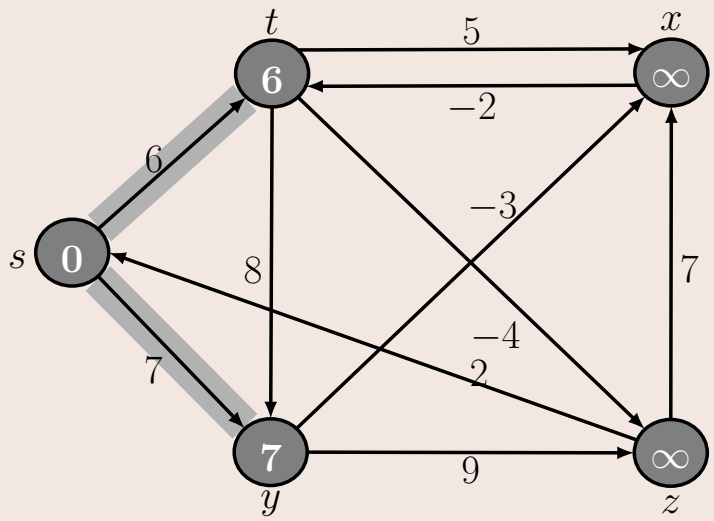
Algoritmo Bellman-Ford

Implementado con *heap* binario

Algorithm Bellman-Ford

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
 1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
 2: for i ← 1 to |G.V| − 1 do
 3:     for cada arista (u, v) ∈ G.E do
 4:         RELAJACIÓN(u, v, w)
 5:     end for
 6: end for
 7: for cada arista (u, v) ∈ G.E do
 8:     if v.d > u.d + w(u, v) then
 9:         return false
10:     end if
11: end for
12: return true
```



Caminos Mínimos

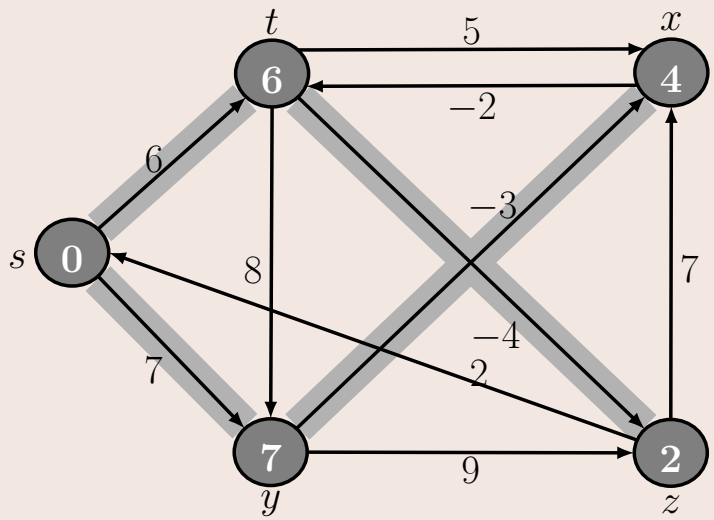
Algoritmo Bellman-Ford

Implementado con *heap* binario

Algorithm Bellman-Ford

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
 1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
 2: for i ← 1 to |G.V| − 1 do
 3:     for cada arista (u, v) ∈ G.E do
 4:         RELAJACIÓN(u, v, w)
 5:     end for
 6: end for
 7: for cada arista (u, v) ∈ G.E do
 8:     if v.d > u.d + w(u, v) then
 9:         return false
10:     end if
11: end for
12: return true
```



Caminos Mínimos

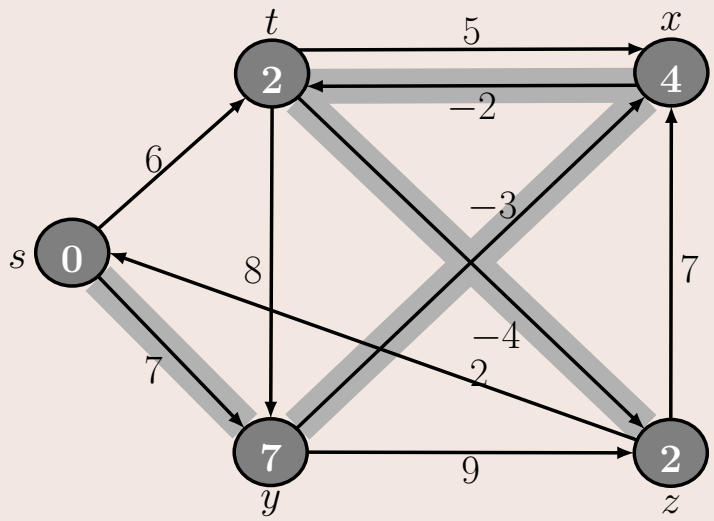
Algoritmo Bellman-Ford

Implementado con *heap* binario

Algorithm Bellman-Ford

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
 1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
 2: for i ← 1 to |G.V| − 1 do
 3:     for cada arista (u, v) ∈ G.E do
 4:         RELAJACIÓN(u, v, w)
 5:     end for
 6: end for
 7: for cada arista (u, v) ∈ G.E do
 8:     if v.d > u.d + w(u, v) then
 9:         return false
10:     end if
11: end for
12: return true
```



Caminos Mínimos

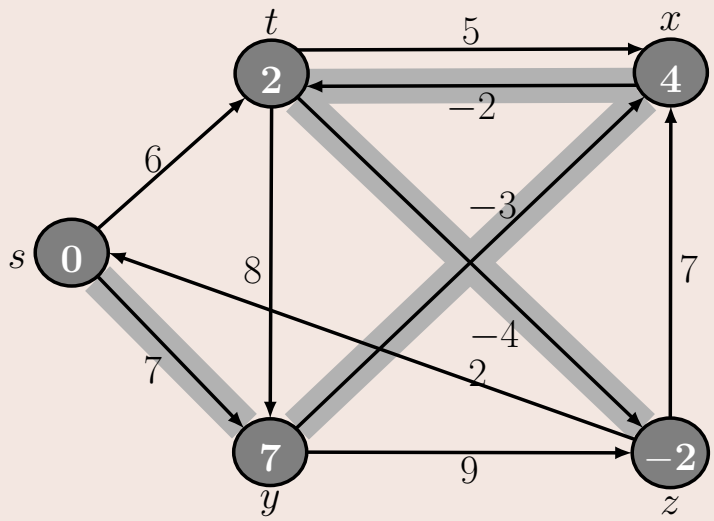
Algoritmo Bellman-Ford

Implementado con *heap* binario

Algorithm Bellman-Ford

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
 1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
 2: for i ← 1 to |G.V| − 1 do
 3:     for cada arista (u, v) ∈ G.E do
 4:         RELAJACIÓN(u, v, w)
 5:     end for
 6: end for
 7: for cada arista (u, v) ∈ G.E do
 8:     if v.d > u.d + w(u, v) then
 9:         return false
10:     end if
11: end for
12: return true
```



Caminos Mínimos

Algoritmo Bellman-Ford

- El Algoritmo de Bellman-Ford corre en un tiempo $O(|V||E|)$ ya que la inicialización toma un tiempo $\Theta(|V|)$.
- Cada uno de los $|V| - 1$ pasos sobre las aristas toma un tiempo $\Theta(|E|)$.
- En tanto, el último ciclo *for* toma un tiempo $O(|E|)$.

Caminos Mínimos

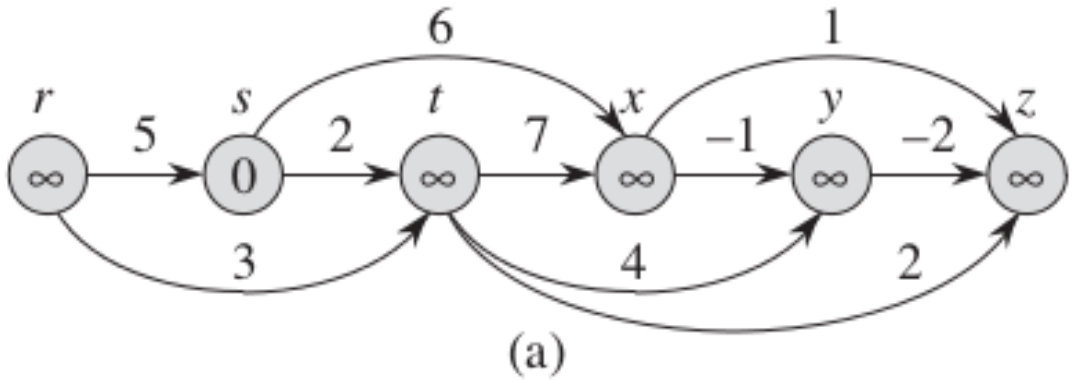
Algoritmo para GDA

- Cuando se relajan las aristas de un GDA (grafo dirigido acíclico) con peso $G = (V, E)$ de acuerdo al orden topológico de sus vértices, se puede calcular caminos más cortos desde un origen a muchos destinos en un tiempo $\Theta(|V| + |E|)$.

- Los caminos más cortos están siempre bien definidos en un GDA, ya que si existen aristas con peso negativo, por definición no pueden existir ciclos.

Caminos Mínimos

Algoritmo para GDA



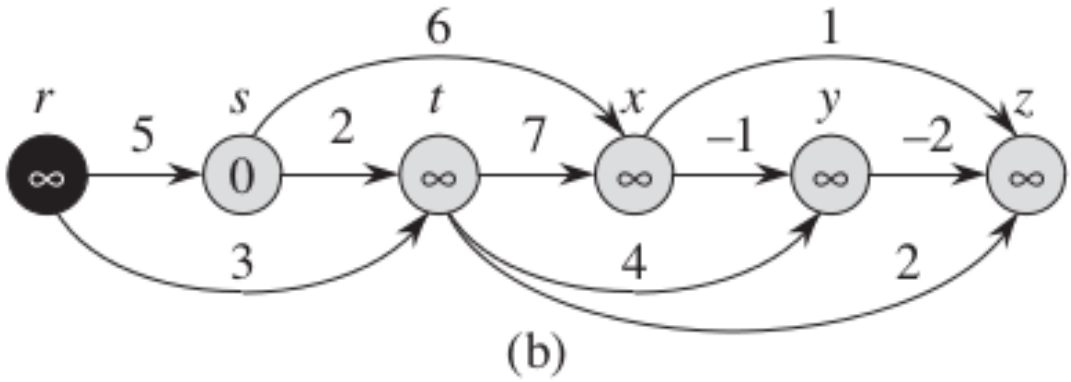
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



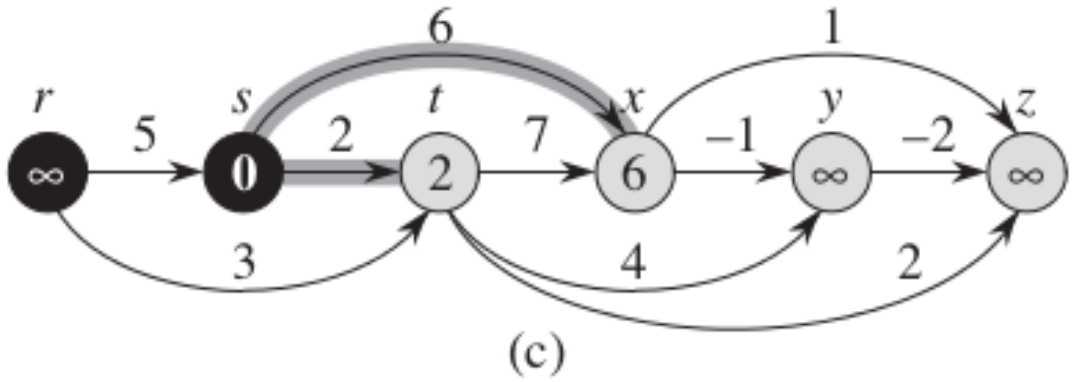
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



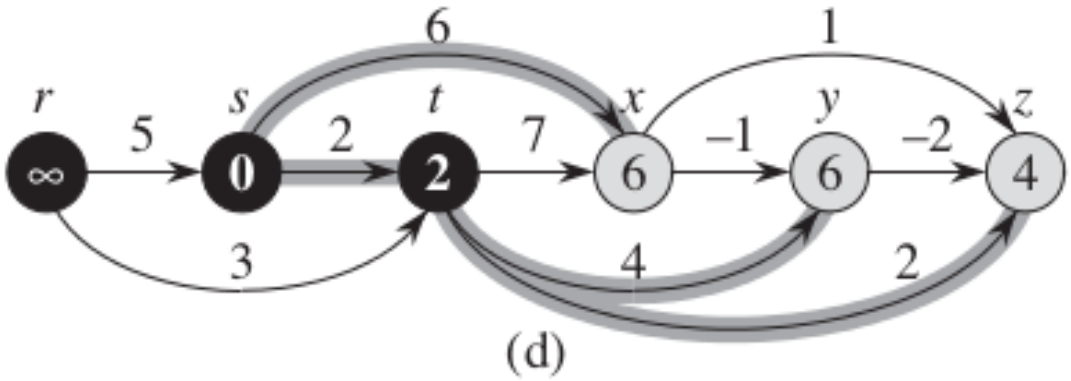
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



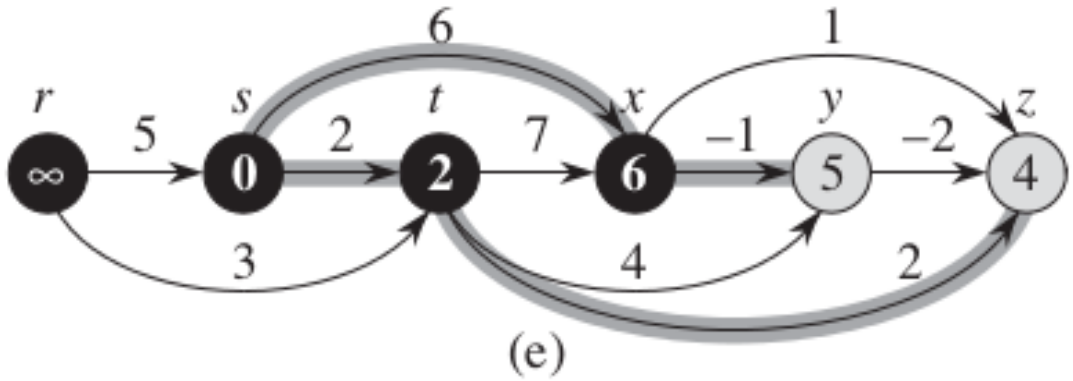
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



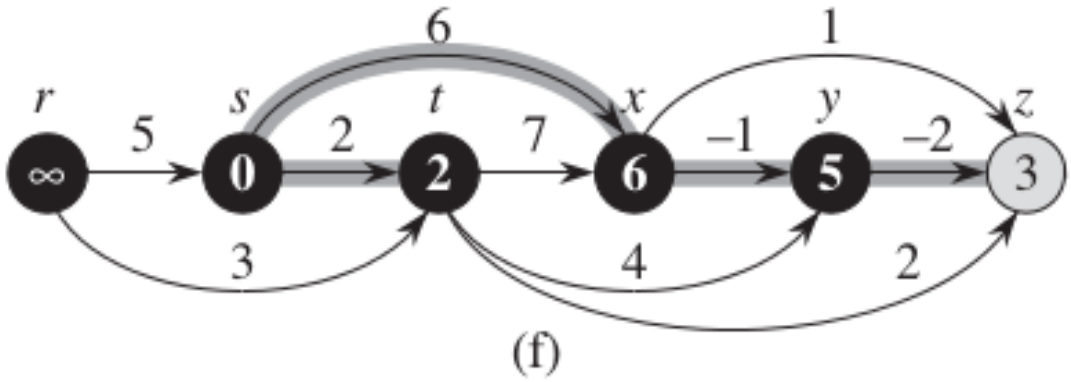
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



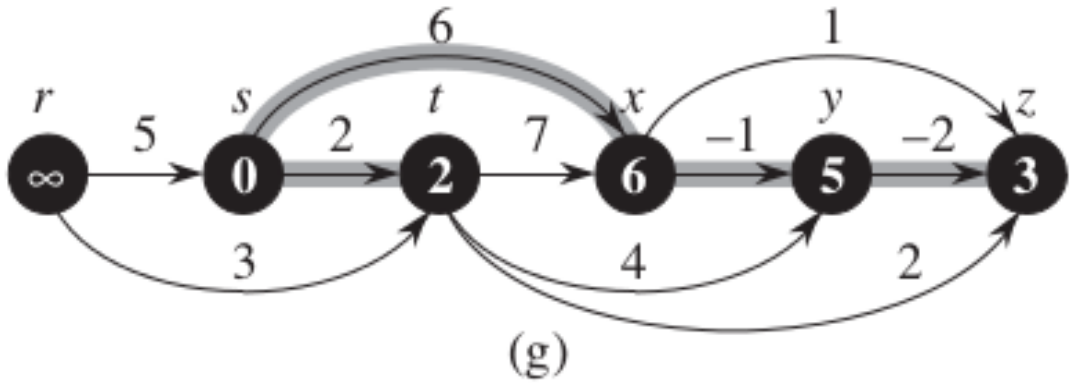
Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA



Algorithm DAG-Camino-Más-Corto

Require: G: Grafo, w: peso de las aristas s: vértice origen.

```
1: Ordenar topológicamente los vértice de G
2: INICIALIZAR-ÚNICO-ORIGEN(G, w, s)
3: for cada vértice u, tomada en orden topológicamente do
4:     for cada vértice v ∈ G.Adyacente(u) do
5:         RELAJACIÓN(u, v, w)
6:     end for
7: end for
```

Caminos Mínimos

Algoritmo para GDA

- El orden topológico puede ser realizado en un tiempo $\Theta(V + E)$.
- El llamado a *Inicializar-Único-Origen* toma un tiempo $\Theta(V)$.
- El ciclo interno hace $|E|$ iteraciones que toman un tiempo $\Theta(1)$ cada una, mientras que el ciclo externo hace una sola iteración por vértice.

Por lo tanto el tiempo total de ejecución de este algoritmo es $\Theta(|V| + |E|)$.

Caminos Mínimos

Algoritmo de Dijkstra

El Algoritmo de Dijkstra resuelve el problema de caminos más cortos desde un origen a muchos destinos en un grafo dirigido con peso $G = (V, E)$ para el caso en el cual todas las aristas tienen un peso no negativo.

- *Heap* binario: Tiene una eficiencia:

$$O((|V| + |E|) \log |V|)$$

.

- *Heap-Fibonacci*: Tiene una eficiencia:

$$O(|V| \log |V| + |E|)$$

.

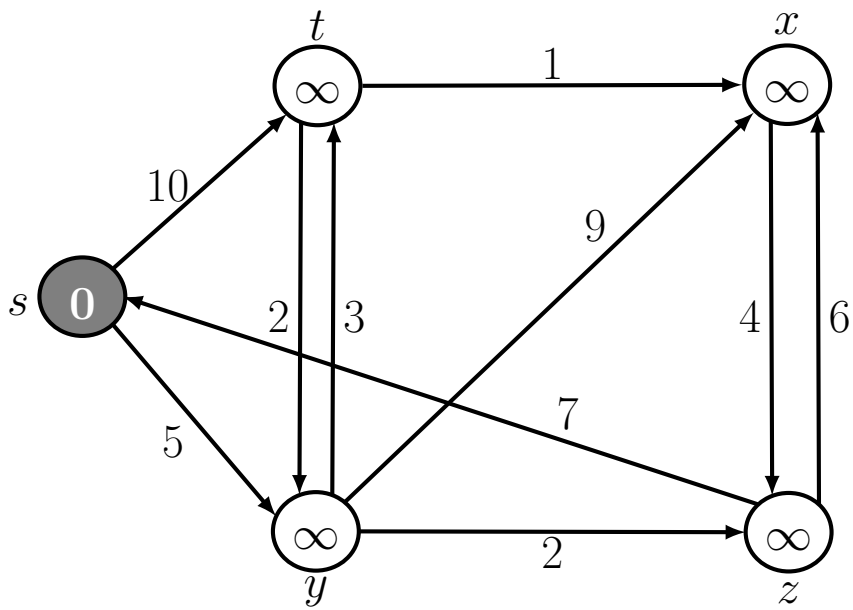
Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```



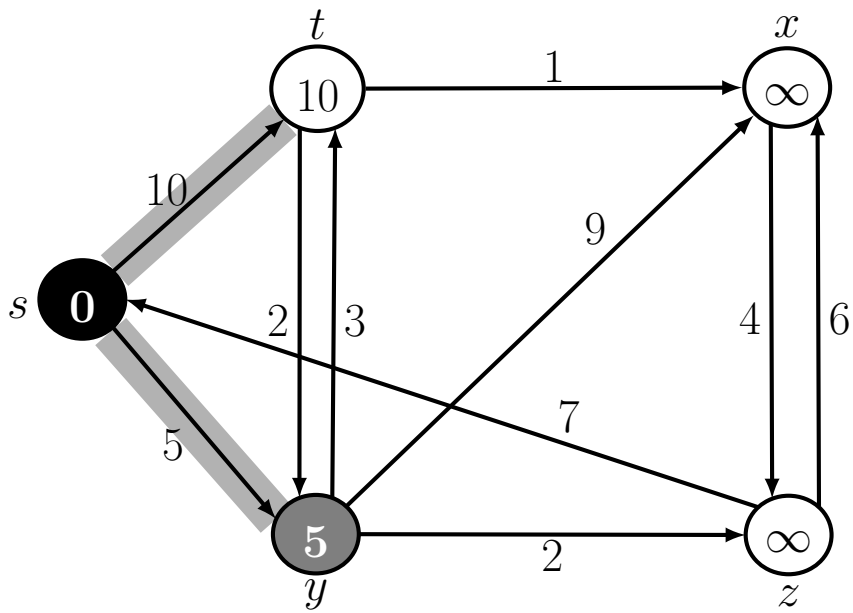
Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```



Universidad Andrés Bello

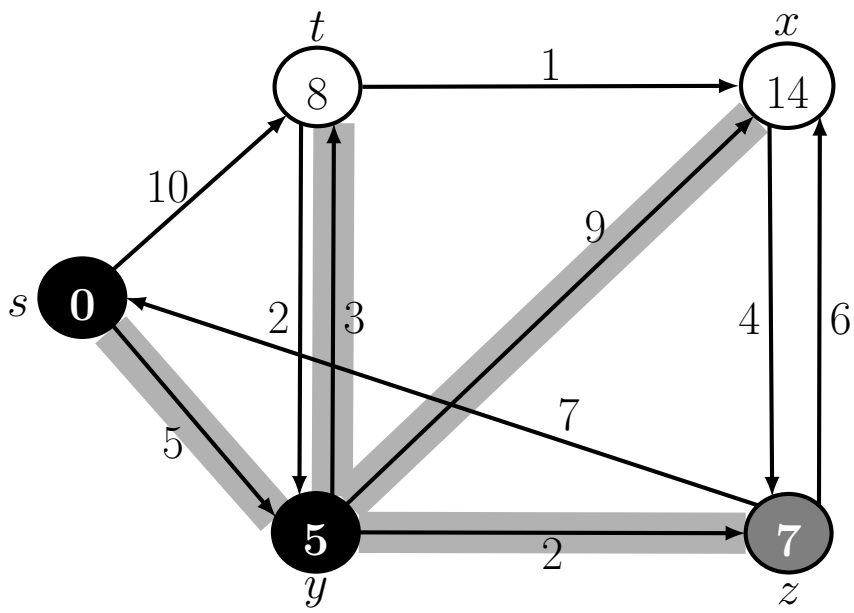
Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```



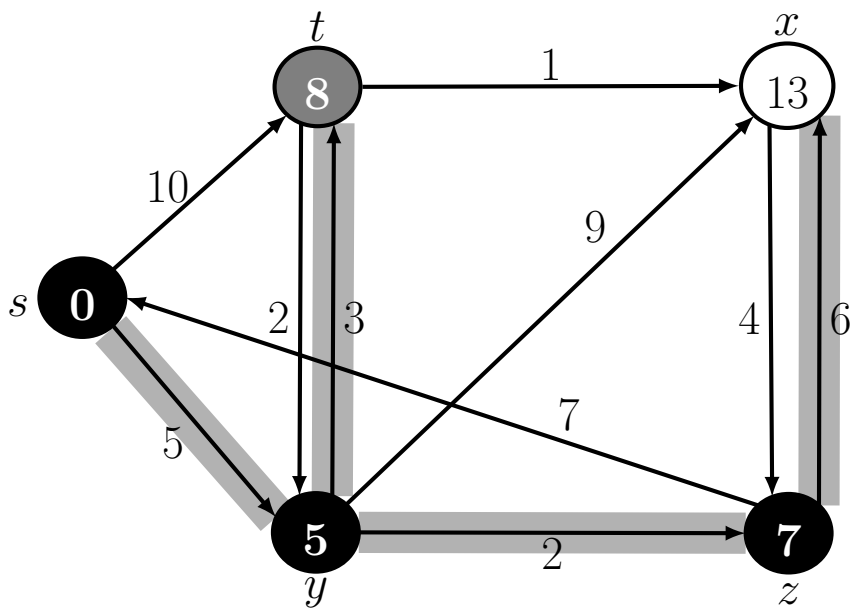
Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```



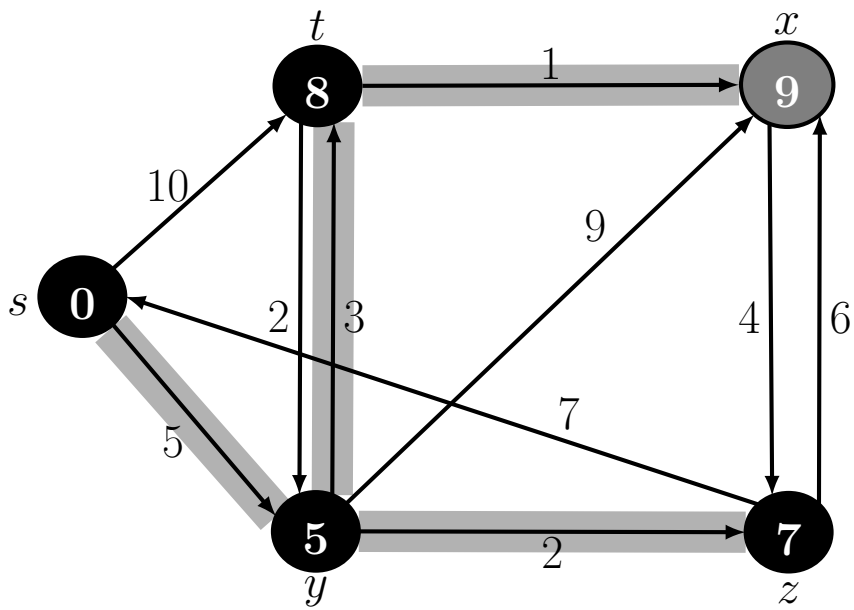
Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```



Caminos Mínimos

Algoritmo de Dijkstra

Algorithm Dijkstra

Require: G: Grafo, w: peso de las aristas, s: vértice origen.

```
1: INICIALIZAR-ÚNICO-ORIGEN(G, s)
2: S ← ∅
3: Q ← G.V
4: while Q ≠ ∅ do
5:     u ←EXTRAER-MIN(Q)
6:     S ∪ {u}
7:     for cada vértice v ∈ G.Adyacente(u) do
8:         RELAJACIÓN(u, v, w)
9:     end for
10: end while
```